VŠB - Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

Simulace propojení dvou paralelních portů

Simulation of Paralel Port Connection

2010 Martin Vlček

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Vlček**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Simulace propojení dvou paralelních portů
Simulation of Paralel Port Connection

Zásady pro vypracování:

Navrhněte simulaci propojení dvou paralelních portů pro potřeby ladění programů komunikujících přes LPT mezi dvěma počítači. Zvolte vhodnou meziprocesní komunikaci pro OS Linux a simulaci navrhněte tak, aby oba procesy využívaly společný virtuální čas.

1. Seznamte se s požadavky na synchronní a asynchronní komunikaci ve cvičeních předmětu ARP.
2. Zjistěte, jaké možnosti nabízí OS pro řízení a programování časové osy s rozlišením pod 1ms. Např. RTC, HPET, HRT a pod.
3. Navrhněte programovou knihovnu, která bude simulovat 6 základních funkcí pro Syn/ASyn komunikaci.
4. Otestujte stabilitu, spolehlivost a odolnost navrženého řešení.
5. Zvažte možnosti přenosu navrženého řešení do OS s WIN32 API.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Petr Olivka**

Datum zadání: 20.11.2009

Datum odevzdání: 07.05.2010

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr.Ing. Eduard Sojka
vedoucí katedry

prof. Ing. Ivo Vondrák, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských/magisterských programech VŠB-TU Ostrava.

V Ostravě dne 7. května 2010 .....

Rád bych na tomto místě poděkoval vedoucímu bakalářské práce ing Petru Olivkovi za pomoc při řešení problémů.

Abstrakt

Tato práce se zaměřuje na vytvoření virtuálního propojení dvou počítačů. Vytvoření aplikace poskytující možnost simulovat arytmickou nebo synchronní komunikaci. Dále pojednává o možnostech časovačů v operačním systému potřebných k této aplikaci.

Klíčová slova: paralelní port, simulace propojení dvou pc, časovače v linuxu, meziprocesní komunikace

Abstract

This work aims to create a virtual connection of two computers. Creating an application that provides the possibility to simulate arrhythmic or synchronous. Further discusses about posibilities of timers in operating system which are needed in this application.

Keywords: parallel port, simulation of connection between two PCs, timers in linux, inter-process communication

Obsah

1. Úvod

V předmětu architektura počítačů, který se vyučuje na fakultě elektrotechniky a informatiky pro studenty druhého ročníku bakalářského studia, se studenti ve cvičeních seznamují s možnostmi připojení dalších zařízení k počítači nebo s programováním jednoduchých komunikačních aplikací mezi dvěma počítači. Jedním z těchto cvičení je nepotvrzovaná komunikace s periferiemi. V tomto cvičení si studenti nejprve zjistí něco o možnostech synchronní a arytmické komunikace dvou počítačů. Pak si vyzkouší napsat jednoduché aplikace pro tuto komunikaci. Nejdříve aplikaci, která posílá nebo přijímá data pomocí arytmického přenosu dat. Potom stejný program se synchronním přenosem dat. A nakonec se pokusí vytvořit jednoduchou aplikaci, která bude posílat data obousměrně. Kdy nejprve vytvoří jednosměrné posílání dat a potom spojením vysílací a přijímací části, vytvoří obousměrnou komunikaci.

V dnešní době již počítače nebývají vybaveny paralelním portem a je potřeba tento port nějak simulovat. Jednou z možností je využít jiného portu v počítači. A nebo vytvoření virtuálního spojení. Kde spojení není tvořeno reálně, ale je pouze uměle vytvořeno mezi dvěma procesy. Tato virtualizace musí poskytovat všechny možnosti, které by poskytovalo opravdové propojení přes paralelní port. Dále musí být implementována tak, aby bylo možno pouze přepsat stávající metody, které se v cvičení používají. Jde o 4 základní metody *get_data*, *set_data*, *get_ctrl* a *set_ctrl*. A pak implementace funkce *delay*, která slouží k pozastavení procesu a funkcí klávesnice. Zde by mohl nastat problém, protože funkce *delay* musí být implementována tak, aby v případě, že čekají oba procesy najednou, ubíhal pro oba stejný čas. Použití, kdy by si každý proces počítal vlastní čas není možné. Procesy totiž nemají vždy možnost běžet stejný čas, protože by se jejich hodnoty dříve nebo později začaly rozcházet a to by způsobilo problémy arytmické komunikace. Je tedy potřeba najít nějaký způsob časování, který by mohly použít oba procesy najednou, přičemž by zároveň příliš nezatěžoval procesor a dokázal by při tom počítat s dostatečně velkým časovým rozlišením.

Tato aplikace bude z důvodu jednodušší iplementace napsána pro linux. V další kapitole bude návrh možností přenositelnosti této aplikace pro operační systém windows, kde budou zmíněny dvě možnosti. Kompilace a spuštění ve virtuálním prostředí cygwin, které emuluje systém unix pod windows. A v druhé části kde se pokusíme rozebrat jak by se musely nahradit části, které by v operačním systému windows nefungovaly.

Nakonec provedeme testování celé aplikace a zhodnotíme jestli je možno tuto aplikaci nasadit k použití v daném předmětu.

2. Synchronní a asynchronní komunikace

2.1. LPT port

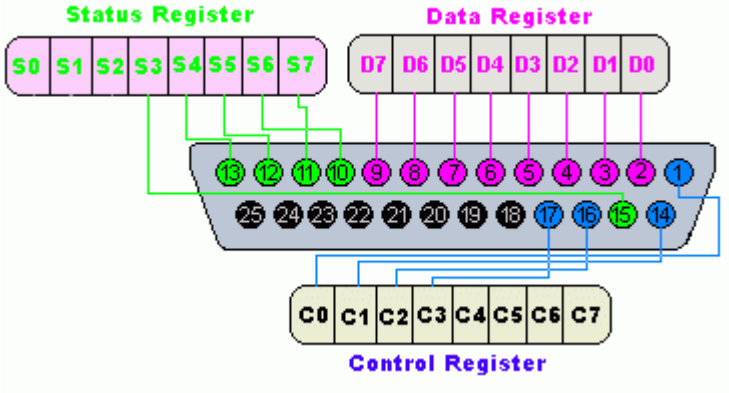
Paralelní port je rozhraní sloužící k obousměrné paralelní komunikaci, i když z počátku byl vymyšlen k jednosměrné komunikaci. Paralelní komunikace znamená, že data nejsou přenášeny po jednotlivých bitech, ale po celých bytech nebo slovech s větší datovou šířkou. Paralelním portem bývá vybavena většina osobních počítačů, i když dnes již ustupují oblíbenějšímu USB (Universal Serial Bus). Tento port původně sloužil především ke komunikaci s tiskárnou. K tomuto účelu se používalo rozhraní Centronics, které bylo donedávna možno najít na prakticky všech tiskárnách. Porovnání paralelního portu a rozhraní Centronix nalezneme na obrázku 1. Později se rozšířilo využití tohoto portu. To bylo způsobeno jednoduchou možností připojit k portu vlastní zařízeni a také způsob ovládání paralelního portu nepředstavuje ani pro začínající programátory žádný větší problém. Většina počítačů byla vybavena většinou jedním nebo dvěma paralelními porty s tradičním nastavením adres portů: LPT1: I/O port 0x378, IRQ 7 a LPT2: I/O port 0x278, IRQ 5. I v případě, že vaše PC nevlastní tento port je možné jej získat pomocí zásuvné karty pro PCI či PCI Express.

Paralelní porty lze spolehlivě propojit na vzdálenost 2 metrů. Avšak v případě použití správného stínění, kdy je každý datový vodič krytý od sousedního zemnícím vodičem, jsme schopni komunikovat až na vzdálenost 5 metrů.

Obrázek 1 Konektor paralelního portu a rozhraní Centronics. [1]

Pro paralelní port se většinou používá 25 pinový konektor. Ve standardu IEEE 1284 je definován také 36 pinový port podobný Centronics ovšem menších rozměrů, ten se obvykle vyskytuje na tiskárnách a jiných periferních zařízeních.

Standardní paralelní port obsahuje 4 skupiny vodičů. V první skupině se nachází 8 datových vodičů D0 až D7 umístěných na pinech 2 až 9. Tento počet byl použit proto, že se v počátcích používala pouze jednosměrná komunikace a dalo se tím pádem přenést najednou celý byte. V další skupině pak 4 stavové výstupní a ve třetí skupině 5 vstupních vodičů. Ke čtvrté skupině se připojují zemnící vodiče, kterých je 8 stejně jako datových vodičů.



Obrázek 2 Přiřazení jednotlivých registru pinům portu. [2]

Rozdělení vodičů je vidět na obrázku 2.Růžové piny zobrazují datové vodiče, modré stavové výstupní, zelené stavové vstupní a černé zemnící

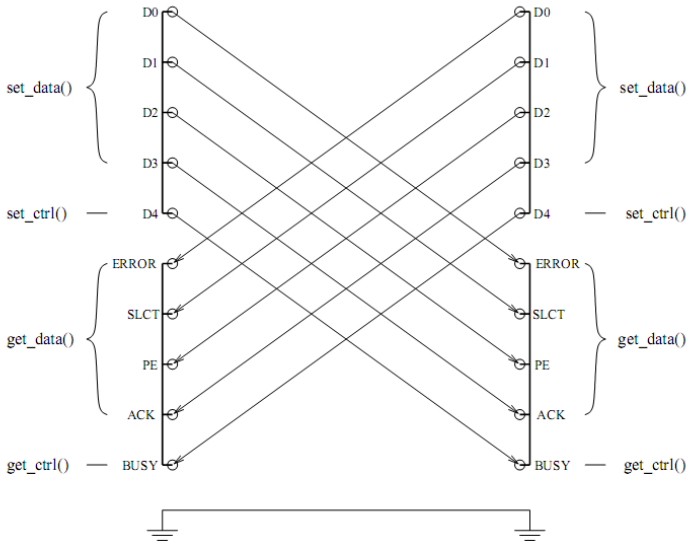
Při obousměrné komunikaci jsou použity datové vodiče, které se rozdělí na 4 vstupní a 4 výstupní pro každou stranu. Z toho vyplývá, že při posílaní dat musí odesilatel dělit data na „půl-byty“, které je schopný najednou odeslat.

2.2. Arytmická (asynchronní) komunikace

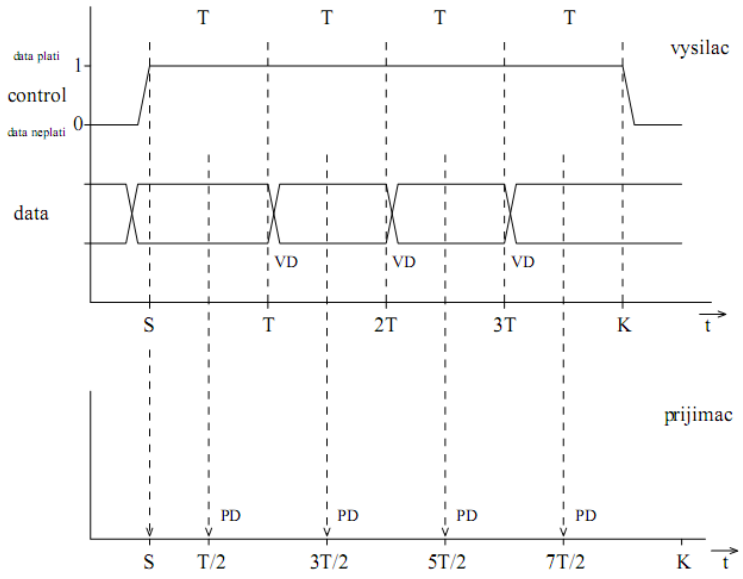
U arytmické komunikace může dojít k začátku přenosu kdykoliv, proto se ji říká asynchronní. K synchronizaci dochází pouze na začátku vysílaní a poté již obě strany používají vlastní hodiny. Tato komunikace je schopná běžet pouze omezenou dobu, protože po určité době přestane být synchronizovaná a přestala by fungovat korektně. Výhodou je, že se v průběhu přenosu nemusí provádět synchronizace, což má za důsledek navýšení rychlosti oproti synchronní komunikaci. Na obrázku 3 můžeme vidět propojení jednotlivých datových vodičů.

Vysílač nejprve nastaví data a poté nastaví *control* na 1, což znamená pro přijímač, že začala komunikace. Vysílač mění data vždy po uplynutí časové periody T. Přijímač může tato data bezpečně přijmout pouze uprostřed této periody. Na začátku tedy počká polovinu této periody a přečte data. Pak čeká vždy celou periodu, aby se dostal zase do půlky periody vysílače, kde může opět přečíst data. To se děje dokud vysílač nezmění *control* na 0. Pak přenos dat končí. Schéma této komunikace je zobrazeno na obrázku 4.

Tato komunikace potřebuje na obou stranách pokud možno co nejpřesnější hodiny, aby komunikace vydržela co nejdéle, nebo alespoň po dobu nutnou k přenesení námi potřebného množství dat. Programátorskému řešení tohoto problému mezi dvěma procesy se věnuji v další kapitole.



Obrázek 3 Propojení dvou počítačů přes paralelní port. [3]

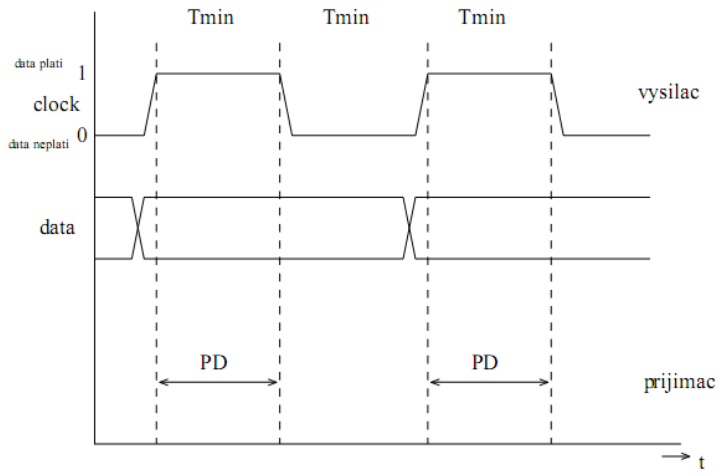


Obrázek 4 Arytmická(asyncrhonní) komunikace schéma průběhu. [3]

2.3. Synchronní komunikace

Naopak synchronní komunikace je řízena hodinovým signálem (clock). To znamená, že vysílač přenáší hodinový signál k přijímači. V našem případě je hodinový signál přenášen po samostatné lince, což je nejjednodušší způsob. Další možností je zkombinovat hodinový signál s datovým signálem, nebo umožnit příjemci, aby se synchronizoval ze samotných dat.

Přijímač čeká a neustálé testuje control, dokud není nastaven je nastaven na 1. Vysílač nejprve nastaví data a pak nastaví control na 1, tím dá vědět přijímači, že může číst data. Ten při neustálém testování tuto změnu okamžitě zaregistruje a data si přečte. Vysílač musí nechat data i *control* nastavené po určitou dobu T_{min}. Aby si přijímač stihl přečíst data. Po uběhnutí této doby změní vysílač hodnotu *control* zpátky na 0. Přijímač se vrátí na začátek a začne testovat *control* dokud nebude roven 1. Vysílač může znovu nastavit data a zahájit znovu komunikaci. Schéma je zobrazeno na obrázku 5.



Obrázek 5 Synchronní komunikace schéma průběhu. [3]

Obě tyto komunikace jsou nepotvrzovány. To znamená, že přijímač nijak nepotvrzuje vysílači přijetí dat. Takže pokud by se stalo, že by se přijímač zablokoval nebo by nastala chyba spojení, vysílač to nijak nepozná a vysílá dál v domnění, že je vše v pořádku.

3. Řízení a programování časové osy s rozlišením pod 1 ms v OS Linux

Kvůli potřebě počítat reálný čas pro dva procesy je potřeba najít správný zdroj časování s rozlišením pod 1 ms. Není možné použít klasické zpoždění v každém procesu zvlášť, protože každý proces nemusí dostávat stejně dlouhý čas od procesoru. Z toho vyplývá, že by synchronizace v arytmickém přenosu nemohla být zaručena. Takže musíme vytvořit časovač, který bude hlídat čas pro oba procesy. V ideálním případě bude počítat čas po desetině minimální hodnoty, kterou může čekat každý proces, což je v našem případě 1 ms. Časovač začne počítat vlastní čas, když ho některý proces požádá. Pokud se připojí další proces k čekání, tak musí jeho čas zarovnat k již rozpočítanému času. Rozlišení pod 1 ms nám v linuxu poskytuje funkce *usleep()* a nebo knihovna linux/rtc.h.

3.1. RTC

Real-time clock je knihovna linuxu, která se stará o obsluhu hardwarových hodin, tyto hodiny se nesmí zaměňovat se systémovými hodinami. Rozdíl mezi nimi je ten, že RTC mají vlastní zdroj napájení a běží i ve chvíli kdy je počítač odpojen od elektrické sítě. Linuxové jádro RTC používá pouze při startu systému k inicializaci systémových hodin.

RTC je schopno také kromě podávání informace o čase, generovat přerušení. Frekvence tohoto přerušení lze nastavit na jakoukoliv mocninu dvou v rozsahu od 2 Hz do 8192 Hz. Z čehož vyplývá, že pro náš časovač by bylo možné tuto funkci použít.

Zařízení může být otevřeno pouze jedním procesem. Ve výpisu 1 můžeme vidět ukázku použití příkazu *open*.

```
int open("/dev/rtc", 0_RDONLY);
```

Výpis 1 Otevření RTC.

Příkaz ioctl slouží k nastavení požadavku pro RTC. My používáme požadavek RTC_IRQP_SET, který nastaví frekvenci přerušení a RTC_AIE_ON, RTC_AIE_OFF, které povolí a na konci zakážou přerušení. To je vidět ve výpisu 2.

```
int file_handler open("/dev/rtc", 0_RDONLY);

int tmp = 4096;

int ioctl( file_handler, RTC_IRQP_SET ,tmp);
```

Výpis 2 Volání požadavku k RTC.

Výpis 3 prezentuje použití časovače pomocí RTC. Toto řešení bylo ovšem nakonec nebylo použito, neboť nastavení frekvence je omezeno. Frekvence nad 64 Hz je schopen nastavovat pouze administrátor. Což by způsobilo omezení naší aplikace.

```
///otevření připojení k RTC
fd = open("/dev/rtc0", O_RDONLY);
int tmp=4096;
///nastavení frekvence přerušení
retval = ioctl(fd, RTC_IRQP_SET, tmp);
///povolení přerušení
if (retval == -1)
{
      perror("RTC_PIE_ON ioctl");
      exit(errno);
}
///čtení přerušení 20krat pak konec
for (i=1; i<21; i++)
{
      retval = read(fd, &data, sizeof(unsigned long));
      if (retval == -1)
      {
            perror("read");
            exit(errno);
      }
      fprintf(stderr, " %d",i);
      fflush(stderr);
}
///zakázaní přerušení
retval = ioctl(fd, RTC_PIE_OFF, 0);
if (retval == -1)
{
      perror("RTC_PIE_OFF ioctl");
      exit(errno);
}
///na konci uzavřeme spojení s RTC
close(fd);
```

Výpis 3 Jednoduchý časovač pomocí RTC.

3.2. Funkce usleep( )

Druhé jednodušší řešení slouží v použití funkce *usleep()*. Funkce sleep a usleep slouží k uspání procesu nebo vlákna na určitou dobu, kde u funkce sleep se nastavuje doba v sekundách a u usleep v mikrosekundách. Z toho vyplývá, že funkce usleep je pro nás naprosto vyhovující, protože nám naprosto stačí počítat po desetinách milisekund.

Toto použití se muselo nejdříve otestovat jestli je možné ho použít z důvodu zatížení systém. Pokud tento časovač neustále poběží tak, aby příliš nezatěžoval procesor a také jestli bude zvládat provádět dostatečný počet cyklů za námi potřebný čas. Což jsme vyzkoušeli pomocí jednoduchého programu, ve kterém běžel tento časovač v několika vláknech najednou. Test

dopadl dobře. Časovač byl schopen i v případě až 7 vláken provádět na každém přes 5000 cyklů za 1 sekundu, což je pro náš účel naprosto dostačující a zatížení procesoru také nebylo veliké.

```
void *mojeVlakno(void *par)
 {

      int *i = ( int * ) par;
      while ( 1 )
      {
            usleep(100);
            (*i)++;
      }
 }


 int main( int *an, char **a )
 {
       pthread_t thread_id;
       void* thread_status;

       int N = atoi( a[ 1 ] );
       int pole[ N ];
       bzero( pole, N * sizeof( int ) ) ;
       for ( int i = 0; i < N; i++ )
       pthread_create( &thread_id , NULL , mojeVlakno , pole + i );
       while ( 1 )
       {
            for ( int i = 0; i < N; i++ )
            printf("%d ", pole[ i ] );
            printf( "\n" );
            sleep(1);
       }
       return 0;
 }
```

Výpis 4 Testování použitelnosti funkce *usleep()*

Tento test je vidět ve výpisu 4. Pokud bylo spuštěno 7 vláken, každé z nich bylo schopno provádět více než 5000 cyklů za jednu sekundu. Zatížení procesoru se přitom pohybovalo kolem 15%. Test sem prováděl na notebooku s dvoujádrovým procesorem Pentium Dual-Core T2060 s taktovací frekvencí 1,6 GHz , který dnes patří k pomalejším procesorům. To znamená, že na dnešních počítačích by měl běžet bez problémů.

4. Návrh knihovny pro Syn/ASyn komunikaci

Řešení bylo implementováno v jazyce C++ a bylo vytvořeno pro operační systém linux. Implementace se skládá ze dvou základních částí, a to z virtuálního portu a klienta. V každé části bylo potřeba vytvořit obsloužení základních funkcí. Tyto funkce jsou nastavení a přečtení hodnoty *control*, nastavení a přečtení dat, funkce *delay* a funkce klávesnice. Tato funkce má za úkol pozdržet proces po určitý čas. Pro komunikaci mezi virtuálním portem a klientem jsem použil sokety, kdy virtuální port tvoří server a klienti se k němu připojují.

4.1. Virtuální port

Aplikace využívá vláken k řešení jednotlivých úkolů. Pro výpis informací na terminál se používá metoda zpráva, která rozlišuje typ zprávy. Jsou rozlišovány informační zprávy, zprávy v režimu ladění a chybové zprávy. Technologii vypisování zpráv jsem použil z připravených programů z předmětu Operační systémy.

4.1.1. Hlavní vlákno

Základní vlákno obstarává počáteční inicializaci proměnných, vytvoření soketového páru ke komunikaci mezi vlákny, vytvoření a spuštění ostatních vláken, vytvoření soketu pro komunikaci s klienty, obstarává připojování klientů přes soket a připojování těchto klientů k vláknům, které je budou obsluhovat. Vlákno kontroluje jestli je k obslužným vláknům připojen klient. Pokud se připojí nový klient zkontroluje, které z vláken nemá přiřazeného klienta. Pošle klientovi potvrzení o připojení a identifikaci vlákna, ke kterému bude připojen. Probudí obslužné vlákno a pošle mu na kterém soketu má klienta. Ovšem pokud by měly obě vlákna již přiřazeného klienta tak jej nepřipojí a jeho soket uzavře.

4.1.2. Obslužné vlákno

Obslužná vlákna jsou vytvořena dva z jedné funkce. Rozlišují se pouze identifikací soketu ke klientovi a kterou část soketového páru používají. Na začátku se vlákno uspí a čeká na soketového páru určeného pro komunikaci mezi obslužnými vlákny. Zde čeká dokud ho neprobudí hlavní vlákno a nepošle mu, který soket bude obsluhovat. Vlákno si nastaví reference na tento soket a také na globální proměnné do kterých se ukládá hodnota *control* a data. Poté se přesune do nekonečné smyčky ve které zůstane dokud se neodpojí klient nebo neskončí celý proces. V této smyčce se opět uspí ve funkci select a čeká dokud mu klient nepošle nějaký dotaz. Pokud klient uzavře spojení tak vlákno nastaví do globální proměnné, že k němu není nikdo připojen a začne zase čekat až mu hlavní vlákno přidělí dalšího klienta.

Komunikace s klientem probíhá pomocí soketu, kde si navzájem posílají dotazy o délce 8 znaků. Pokud přijde od klienta něco jiného než domluvený signál, tak jej zahodí. Jinak proběhne obsluha tohoto příkazu.

4.2. Popis komunikačního protokolu

Komunikační protokol pro komunikaci s klientem byl zvolen tak, aby pokaždé byla posílána 12 znaková zpráva. Vetšinou prvních 8 znaků určuje druh operace a poslední 4 znaky obsahují posílané data. Pokud by byla zpráva kratší je doplněna číslicemi. Délka zpráv zasílaných mezi vlákny a časovačem není určena.

4.2.1. Hlavní vlákno – klient

Ze strany hlavního vlákna

portXXXXXXXX	hlavní vlákno zasílá potvrzení o přijetí procesu a informací ke kterému vláknu bude proces připojen, v posledních 8 znacích je tato identifikace

4.2.2. Obslužné vlákno – klient

Ze strany klienta:

set_ctrl000X	klient posílá hodnotu *control* 1 nebo 0
set_data00XX	klient posílá hodnotu dat v hexadecimálním kódu
getstate0000	klient žádá o poslání stavu portu
getstateXXXX	klient žádá o poslání stavu portu za určitý čas, čas je zakódován v posledních 4 znacích
delay_msXXXX	klient žádá o čekání, čas je zakódován v posledních 4 znacích

Ze strany obslužného vlákna:

set_ctrl000X	klientovi je poslána potvrzovací zpráva o přijetí nastavení *controlu*
set_data00XX	klientovi je poslána potvrzovací zpráva o přijetí nastavení dat
getstate0XXX	klientovi je zaslán stav portu v posledních dvou znacích jsou uložena data a v třetím znaku od konce je uložen *cotrol*
over11111111	je poslán konec čekání klientovi

4.2.3. Obslužné vlákno 1 – 2

Vlákna si akorát navzájem přepošlou zprávu o změně dat nebo controlu.

4.2.4. Obslužné vlákno – časovač

Ze strany obslužného vlákna:

start	probuzení vlákna, požadavek k čekání

Ze strany časovače

OVER	konec čekání pro vlákno

4.3. Rozšifrování zpráv od klienta :

set_ctrl

Klient chce nastavit *control*. V posledním znaku poslané zprávy je zakódována tato hodnota. Vlákno ji tedy uloží do globální proměnné a pošle zprávu druhému vláknu o změně hodnoty. Nakonec pošle zpět potvrzovací zprávu, že dostal a zpracoval zprávu od klienta. Zdrojový kód je vidět ve výpisu 4.

```
if(!strncasecmp(buf,"set_ctrl",8))
{
	ctrl_w[1] = buf[11];
	send( sockets[muj_port], buf, 8, 0);
	///posilam potvrzeni ze sem ctrl prijal

	write(muj_socket,buf,12);
	zprava(ZPR_LADENI,"vlakno%d: klient nastavil ctrl %s. ",muj_port,ctrl_w);
}
```

Výpis 5 Funkce set_ctrl() pro nastavení control.

set_data

Obdobné jako u *set_ctrl*, ale klient nastavuje data, ty jsou uloženy ve 2 posledních znacích poslané jako hexadecimální číslo. Opět se o této akci informuje druhé vlákno a poté je potvrzeno přijetí zprávy klientovi. Zdrojový kód je vidět ve výpisu 5.

```
if(!strncasecmp(buf,"set_data",8))
{
	memmove(data_w,buf+10,2);
	send( sockets[muj_port], buf, 8, 0);
	///posilam potvrzeni ze sem prijal data
	write(muj_socket,buf,12);
	zprava(ZPR_LADENI,"vlakno%d: klient nastavil data %s.",muj_port,data_w);
}
```

Výpis 6 Funkce set_data() pro nastavení dat.

getstate

Klient žádá o poslání stavu portu. To znamená, že chce poslat *control* i data zpět a on již použije pouze tu hodnotu, kterou potřebuje. V posledních 4 znacích jsou buď nastaveny nuly, což znamená, že klientovi pošlu stav portu ihned. To udělá tak že pošle zpět zprávu *getstate*, do dvou předposledních znaků nastaví *control* a do dvou posledních data. Jinak tam může být nastaveno nějaké číslo. To znamená, že klient zjistil, že čte některou hodnotu příliš často. Z důvodu zabezpečení, aby se nezablokoval soket chce před posláním určitou chvíli počkat. Tuto dobu tím pádem nastavil do těch posledních 4 znaků. Ty jsou převedeny do struktury *timeval* a pak zustane čekat ve funkci *select* do vypršení doby. Ten může být přerušen druhým vláknem, pokud byly na druhé straně změněny data nebo control. Po ukončení funkce *selec*t, ať již doběhnutím času nebo zprávou z druhé strany, pošle klientovi data. Zdrojový kód je vidět ve výpisu 6.

```
if(!strncasecmp(buf,"getstate",8))
{
	if(!strncasecmp(buf+8,"0000",4))
	{
		memmove(buf+8,ctrl,2);
		memmove(buf+10,data,2);
		write(muj_socket,buf,12);
		zprava(ZPR_LADENI,"Klient chce poslat stav portu");
	}
	else
	{
		///klient my poslal zpravu ze cte neco prilis
		///casto a tak chce po me at mu odpocitam cas
		///a poslu mu pak data nebo pokud se zmeni driv
		///tak mu je poslu driv
		char cas[5];
		memmove(cas,buf+8,4);
		timeval * my_time;
		timeval time;
		time.tv_usec = atoi( cas )*1000;
		my_time = &time;
		fd_set my_set;
		FD_ZERO( &my_set );
		FD_SET( sockets[muj_port], &my_set );
		///ja se zastavim na selectu a bud dojde cas
		///nebo me probudi zprava z druhe strany portu
		select(sockets[muj_port]+1,&my_set,0,0,my_time);
		///pote poslu klientovy data zpatky
		memmove(buf+8,ctrl,2);
		memmove(buf+10,data,2);
		write(muj_socket,buf,12);
		zprava(ZPR_LADENI,"Poslal jsem klientovy stav portu");
	}
}
```

Výpis 7 Funkce getstate() pro získání stavu portu.

delay_ms

Klient potřebuje počkat určitý čas, tento čas je nastaven v posledních 4 znacích v ms. Tuto hodnotu nastaví vlákno do globální proměnné, kterou čte časovač a pošle se přes soketový pár zpráva časovači s žádostí o čekání. Poté se čeká dokud nepřijde zpráva od časovače, že došlo ke konci čekání a tato zpráva je poslána klientovi. Zdrojový kód je vidět ve výpisu 7.

```
if(!strncasecmp(buf,"delay_ms",8))
{
	///ja si prve rozsifruju jak dlouho chce cekat,
	//pak nastavim cas do moji globalni promenne
	///a poslu casovaci ze chci cekat
	char cas[5];
	memmove(cas,buf+8,4);
	memmove(buf,"start",5);
	cas_v[muj_port] = atoi(cas);
	send(sockets_timer[0],buf,5,0);
	///pote cekam az mi casovac vrati ze je konec cekani
	while(strncasecmp(buf,"OVER",4))
	{
	  recv(sockets[muj_port],buf,5,0);
	}
	///a pak to preposlu klientovi
	zprava(ZPR_LADENI,"vlakno%d: konec cekani pro klienta",muj_port);
	memmove(buf,"over11111111",12);
	write(muj_socket,buf,4);
}
```

Výpis 8 Funkce delay_ms() pro volání časovače.

4.4. Časovač

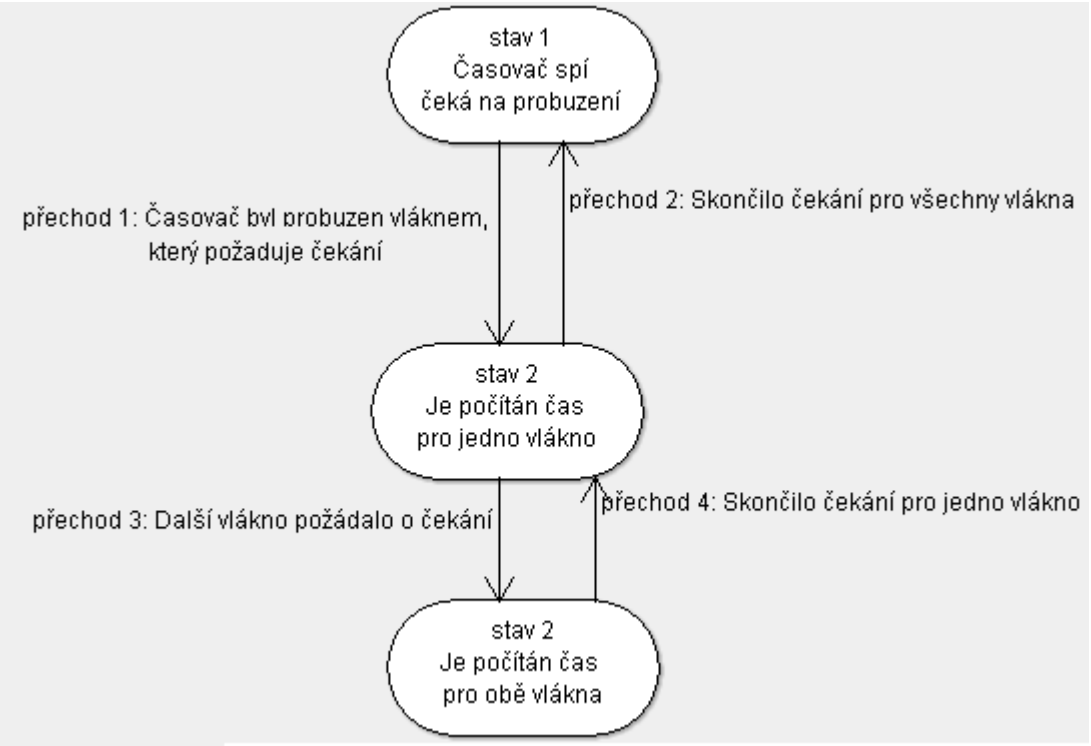
Vlákno s časovačem nabývá 3 základních stavů:

1. Nikdo nechce čekat a spí na selectu, kde čeká na probuzení od obslužných vláken.
2. Počítá čas jednomu vláknu.
3. Počítá čas oběma vláknům.

A má 4 přechody mezi těmito stavy, ty sou také vidět na obrázku 6:

1. Spánek – Počítání času pro jedno vlákno: tento přechod nastane pokud je časovač probuzen některým vláknem a požádán o časování.

2. Počítání času pro jedno vlákno – spánek: časovač skončil s počítáním času jednomu vláknu a není komu počítat další čas tak se uspí.
3. Počítaní času pro jedno vlákno – počítání času pro obě vlákna: časovač již počítá pro jedno vlákno, ale k počítání se připojí další vlákno.
4. Počítaní času pro obě vlákna – počítání času pro jedno vlákno: časovač dopočítal čas jednomu vláknu, ale ještě zbývá jedno vlákno, které chce se musí dopočítat.



Obrázek 6 Stavy časovače a přechody mezi nimi.

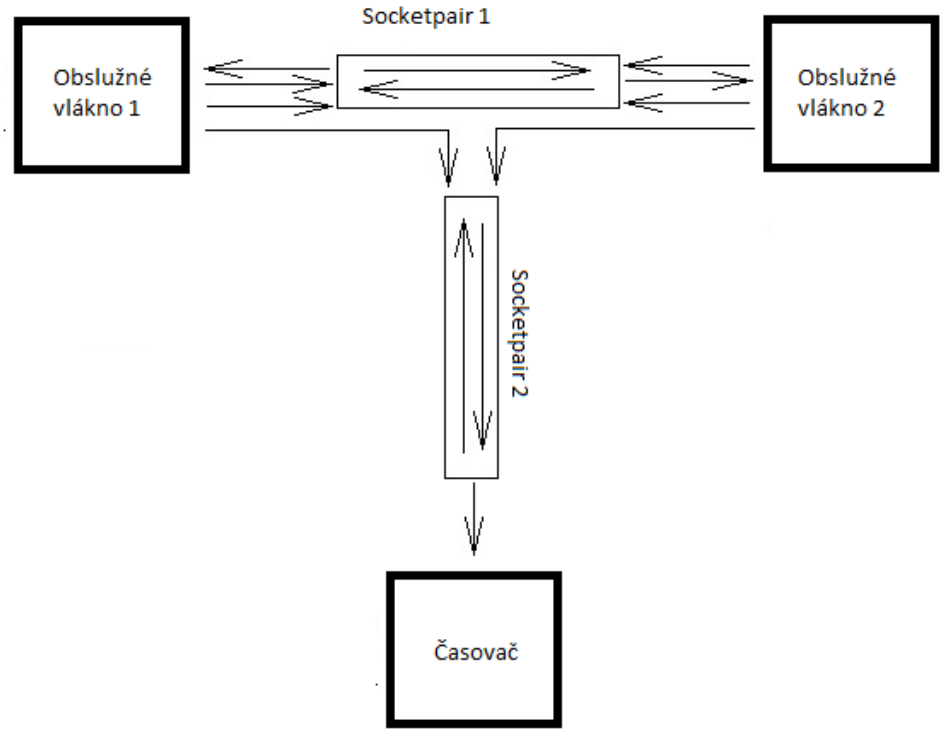
Pro komunikaci s vlákny se používají oba soketové páry. Zprávy časovači posílají vlákna soketovým párem určeným přímo pro tuhle potřebu. Pro zpětné poslání zprávy o konci čekání použije časovač soketový pár, který slouží pro komunikaci mezi vlákny.

Při začátku časování si nastavím koncový čas pro vlákno, které chce čekat a pak začnu pomocí funkce *usleep(100)* počítat čas po desetinách ms. Pokud se v té době připojí další vlákno k výpočtu čekání, k jeho koncovému času přičtu můj momentální čas a počítám dál. Pokud je počítaný čas roven koncovému času některého vlákna je mu zaslána zpráva o končení čekání a jsou vymazány jeho časy. Pokud již nikdo čekání nepotřebuje, časovač se uspí a čeká na nové probuzení. Zdrojový kód je vidět ve výpisu 8.

```
if(FD_ISSET(sockets_timer[1],&read_wait_set))
{
      recv( sockets_timer[1],buf,5,0);
      ///nastavim si koncove stavy a zvetsim jim rozliseni 10x
      cas_v[0] = cas_v[0] * 10;
      cas_v[1] = cas_v[1] * 10;
      ///nastavim si pomocne casy pomoci kterych rozpoznavam
      ///jestli nechce cekat i jine vlakno
      cas_01 = cas_v[0];
      cas_02 = cas_v[1];
      if(!strncasecmp(buf,"start",5))
        ///dokud se oba vysledne casy nerovnaji 0
        while(cas_v[0] != 0 || cas_v[1] != 0)
        {
          ///tohle je moje casovaci jednotka 0,1ms
            usleep(100);
            cas++;
            ///zjistuju jestli se nezmenilo nastaveni
            /// koncoveho casu pro vlakno 1
            if(cas_01 != cas_v[0])
            {
            ///jestli jo tak prenastavim jeho koncovi cas
            /// tak ze k nemu prictu muj aktualni cas
                  cas_v[0] = cas_v[0]*10 + cas;
                  cas_01 = cas_v[0];
            }
            ///zjistuju jestli se nezmenilo nastaveni
            ///koncoveho casu pro vlakno 1
            if(cas_02 != cas_v[1])
            {
                  ///jestli jo tak prenastavim jeho koncovi
                  ///cas tak ze k nemu prictu muj aktualni cas
                  cas_v[1] = cas_v[1]*10 + cas;
                  cas_02 = cas_v[1];
            }
            ///pokud sem dosahl koncoveho casu prvniho vlakna
            /// tak mu poslu zpravu ze skoncilo jeho cekani
            if(cas_v[0] == cas)
            {
                    memmove(buf,"OVER",4);
                    cas_v[0] = 0;
                    cas_01 = cas_v[0];
                    send(sockets[1],buf,5,0);
            }
            ///pokud sem dosahl koncoveho casu druheho vlakna
            ///tak mu poslu zpravu ze skoncilo jeho cekani
            if(cas_v[1] == cas)
            {
                     memmove(buf,"OVER",4);
                     cas_v[1] = 0;
                     cas_02 = cas_v[1];
                     send(sockets[0],buf,5,0);
            }
        }
       ///kdyz nechce nikdo cekat tak si vynuluju cas
       cas = 0;
}
```

Výpis 9 Kód časovače.

Ke komunikaci mezi vlákny a mezi vlákny a časovačem jsou použity dva soketový pár. Soketový pár funguje jeho obousměrná roura. Pokud komunikují vlákna spolu použijí první soketový pár v obou směrech. Pro posílání zprávy časovači použijí vlákna druhý soketový pár. Když chce poslat zprávu časovač některému vláknu, použije opět první soketový pár a pošle zprávu přímo tomu vláknu, kterému potřebuje. Schéma použití je zobrazeno na obrázku 7.



Obrázek 7 Schéma soketový pár mezi vlákny a časovačem.

4.5. Klient

Klient je upravená aplikace, která se nyní používá v předmětu ARP pro komunikaci mezi dvěma počítači přes paralelní port. V této aplikaci jsou zachovány funkce *get_data*/*set_data* a *get_ctrl*/*set_ctrl*. Je připsána funkce delay pro čekání která využívá služeb časovače a ještě jsou vytvořeny funkce pro čtení dat z klávesnice. Funkce kbhit, která zjišťuje jestli nebyla stisknuta klávesa. Tato funkce funguje tak, že je volána funkce select, která čeká na STDIN s dobou čekání nastavenou na nulu. Hodnota, kterou funkce vrátí, určí, jestli klávesa byla stisknuta nebo ne. Dále jsou zde pak funkce getch a getche, které čtou znaky z klávesnice.

Při spuštění je nutno zavolat funkci *init(),* která inicializuje soket a připojí se k serveru, tím je virtuální port. Dále pak inicializační funkce pro čtení z klávesnice. Pak je možno napsat svůj vlastní program pro komunikaci.

Na straně klienta je řešen problém zahlcení soketu neustálými dotazy. Kde je použita funkce gettimeofday, která vrací aktuální čas. Tento čas je zaznamenán při každém volání funkce *get_ctrl*, *get_data* i *kbhit*. Pro každou funkci je počítán vlastní čas. Pokud je některá z těchto funkcí volána příliš často. To znamená, že poslední volání bylo před méně než 10 ms tak je vyřízení tohoto požadavku pozdrženo. To je řešeno pomocnou funkcí *Pselect*, ta je vypsána ve výpisu 9. Nejprve si necháme poslat stav portu a pak zavoláme tuto funkci. Do ní pošleme jako parametr jak dlouho chceme čekat. Tato funkce požádá virtuální port o čekání. Pak nastaví *select* pro čekání na soket a na STDIN. Pokud přijde první odpověď z STDIN tak buď pošle odpověď „true“, což znamená, že byla stisknuta klávesa. Pokud přijde odpověď ze soketu, vrátí hodnotu portu. Jaká odpověď přišla, se řeší v samostatných funkcích *get_data*, *get_ctrl* a *kbhit*. Funkce pak testují jestli jim v odpovědi přišlo to co potřebují. Pokud do funkce kbhit přijde jako odpověď stav portu, je bráno, že klávesa stisknuta nebyla. Pokud se do funkce *get_data* nebo *get_ctrl* vrátí „true“ nebo „false“ použije funkce data portu, které získal na začátku, neboť nepřišla zpráva o změně, jsou brána jako aktuální.

```
FD_SET( STDIN_FILENO, &rs );
FD_SET( sock_server, &rs );
int r = select( STDIN_FILENO + 1, &rs, 0, 0, NULL );
if(FD_ISSET( sock_server,&rs))
{
      read( sock_server, buf, celkova_delka );
      memmove(stav,buf+prvni_cast,druha_cast);
      return stav;
}
else
{
      if(FD_ISSET( STDIN_FILENO, &rs ))
      {
            return "true";
      }
      else
            return "false";
}
return  "false";
```

Výpis 10 Funkce PSelect(), která slouží k pozdržení volání funkci *get_data*, get_ctrl nebo kbhit.

Použití funkce *delay* pro pozdržení procesu v milisekundách. Tato hodnota je funkci posílána v parametru. Funkce nejdříve testuje, jestli nebyla poslána hodnota větší než 9999 ms, což je maximum, které můžeme virtuálnímu portu, z důvodu délky posílané zprávy, kde jsou pro data určené pouze 4 znaky. Pokud je tato hodnota větší rozdělí na několik časových úseků po 5 sekundách a počká nejdříve tyto kratší úseky. A nakonec počká zbytek času, který zbývá. Samotné čekání je tvořeno posláním požadavku na soket se zprávou *delay_ms* a počtem ms, které chce počkat. A poté počká u soketu, až mu virtuální port pošle zprávu o konci čekání. Zdrojový kód je vidět ve výpisu 10.

```
sprintf(casp,"%04d",cas);
sprintf(buf,"delay_ms%s",casp);
write( sock_server, buf, celkova_delka);
while(strncasecmp(buf,"over111111111",celkova_delka))
{
      read( sock_server, buf, celkova_delka );
}
```

Výpis 11 Funkce delay pro pozdržení procesu.

5. Testování

Program bylo potřeba otestovat pro možnosti, které mohou nastat v cvičeních předmětu architektura počítačů. Nejprve jsem testoval arytmickou komunikaci, protože u ní není potřeba testovat spolehlivost časovače. Ten pouze slouží k pozdržení vysílacího procesu a na druhý proces nemá žádný vliv. Takže stačilo pouze napsat jednoduchý program pro posílání a přijímání dat, kde přijímač neustále aktivně kontroluje hodnotu *control* a čeká až ji vysílač nastaví na 1. Ten ovšem nejprve nastaví data. Pak přijímač počká, až bude *control* roven 0 a celý cyklus začne nanovo. Tenhle typ komunikace je velmi jednoduchý a předem se dá říct, že by zde neměl nastat žádný problém. To se potvrdilo také při testování. Žádný problém také nenastal. Jediná věc která se zde dá ovlivnit je minimální perioda po kterou musí nechat vysílač nastavenou hodnotu *control* na 1. To je potřeba, aby přijímač stihl postřehnout tuto změnu a mohl si tím pádem přečíst data. Při testování fungovala komunikace bez problémů do hodnoty 10 ms, při nastavení nižší hodnoty, se jednou za čas stalo, že přijímač nepřečetl některou hodnotu.

U synchronní komunikace bylo potřeba otestovat možnosti časovače. Jestli splňuje požadavky, které byly popsány dříve v kapitole o časovačích. V programu, který se píše ve cvičení se udržuje synchronizace pouze na poslání několika málo znaků. To znamená, že její udržení nemusí být nijak dlouhé. Pouze je potřeba, aby se udrželo po tuto nutnou dobu. Dále jsem časovač testoval na schopnosti udržení synchronizace. Kde se jednalo o test, kdy jsem nechával oba procesy čekat vždy určitý časový úsek několikrát za sebou a sledoval jsem o kolik se na konci tohoto testu liší časy, kdy končilo čekání. Ve většině případů byl rozdíl pouze v desetinách milisekund.

Jinak bylo spojení v obou případech stabilní, problém může akorát nastat pokud někdo ukončí násilně klienta stisknutím CTRL+C nebo ukončí proces příkazem k ukončení procesu. Potom pokud se tak stalo zrovna v případě, kdy probíhala komunikace tak to může způsobit i pád virtuálního portu. Ale to pouze v případě násilného ukončení klienta a jen někdy. Je tedy lepší pokud jsme museli klienta násilně vypnout zkontrolovat jestli ve virtuálním portu nenastal nějaký problém.

6. Přenositelnost do WIN32 API

6.1. Kompilace a spuštění pod emulátorem cygwin

První možností použitelnosti programu ve WIN32 API je emulace linuxového prostředí pomocí programu cygwin. Tento program emuluje prostředí linuxu pod windows s většinou možností co máme v linuxu. Zjednodušení by bylo v tom, že by se nemusel přepisovat zdrojový kód programu. Po kompilaci je vytvořen přímo spustitelný „exe“ soubor, který ovšem musí být spuštěn opět v programu cygwin, neboť používá knihovny cygwinu. Kompilace a spuštění programu proběhne v pořádku. Ovšem problém nastane ve chvíli, kdy se pokusíme použít časovač. Ten bohužel neběží tak jak má, ale jeho zpoždění nefunguje jak má. Při nastavení čekání 200 ms se protáhne doba čekání na přibližně 4 vteřiny. Což je pro naše použití naprosto nepřijatelné. Z toho vyplývá, že kompilace a spuštění v programu cygwin není reálné pro pro použití programu ve windows. To znamená, že jediná možnost je přepsat zdrojový kód tak, aby byl použitelný ve windows.

6.2. Použití soketů ve WIN32 API

Používání soketů ve windows je velmi podobné jako v linuxu. Používá se knihovna s hlavičkovým souborem windows.h. Jediný rozdíl v použití je v tom, že funkce nevrací hodnotu v integer hodnotu přímo v datovém typu soket. Funkce vrací v případě problému INVALID_SOCKET nebo SOCKET_ERROR. Jinak názvy funkcí jsou stejné jako v linuxu a také jejich parametry.

6.3. Možnosti časovačů ve WIN32 API

Použití funkce usleep není ve windows možné, protože tam funguje pouze klasický *Sleep* v sekundách nebo *SleepEx* v milisekundách, což není pro náš časovač použitelné. My potřebujeme něco, co nám dá možnost počítat čas přibližně v desetinách ms.

Jednou z možností časovačů ve windows by mohla být funkce *QueryPerformanceCounter*. Tato funkce nám poskytuje možnosti časování v mikrosekundách, což je naprosto dostačující pro nahrazení funkce usleep. Kde by stačila napsat jednoduchá funkce, která by nahrazovala usleep a nemusel by se měnit celý kód. Tento kód je vidět ve výpisu 11.

Tato funkce má ovšem svůj problém. Za prvé se podle dokumentace může stát, že některý počítač nemusí mít systém pro počítání času ve vysokém rozlišení. A dalším problémem je, že čítač se může výrazně lišit v různých systémech. Tím pádem se může lišit doba kterou čekáme, to by ovšem pro naši aplikaci nedělalo takový problém, když používáme jeden čítač pro oba procesy. Důležité je, že i když tento čas sice nebude přesný bude pro obě vlákna stejný.

```
usleep (long usec)
{
  LARGE_INTEGER lFrequency;
  LARGE_INTEGER lEndTime;
  LARGE_INTEGER lCurTime;
  QueryPerformanceFrequency (&lFrequency);
  if (lFrequency.QuadPart)
  {
        QueryPerformanceCounter (&lEndTime);
      lEndTime.QuadPart += (LONGLONG) usec * lFrequency.QuadPart / 1000000;
      do
      {
              QueryPerformanceCounter (&lCurTime);
          Sleep(0);
        }while (lCurTime.QuadPart < lEndTime.QuadPart);
  }
}
```

Výpis 12 Ukázka kódu, kterým by šlo nahradit funkci usleep ve windows [4]

7. Závěr

Cílem této práce bylo vytvořit aplikaci, která bude schopná nahradit spojení mezi počítači reálným kabelem přes paralelní port k účelu použití ve cvičení předmětu architektura počítačů. V těchto cvičeních je používají studenti k pochopení principu této komunikace.

Implementace jednotlivých částí se zdařila. Podařilo se vyřešit i problém časovače pro dva procesy. Funkčnost klienta i virtuálního portu je v mezích použitelnosti. Žádné větší problémy nebyly objeveny.

Podle analýzy přenositelnosti na WIN32 API se dá prohlásit, že by nebyla obtížná. Není sice možné spouštět tuto aplikaci ve virtuálním prostředí cygwinu ovšem přepsání aplikace by nebylo složité. Stačilo by upravit použití soketů a vytvořit funkci časovače například pomocí funkce *QueryPerformanceCounter*. Ta by se musela ovšem nejprve otestovat například způsobem, kterým byly testovány funkce knihovny RTC a funkce *usleep*. To by mohlo být také budoucím rozšířením tohoto projektu.

Osobním přínosem této práce bylo rozšíření schopností programování meziprocesní komunikace v linuxu, použití časovačů v operačním systému a používání vláken.

8. Literatura

[1] Tišnovský, Pavel, *Paralelní port a rozhraní Centronics*, http://www.root.cz/clanky/paralelni-port-a-rozhrani-centronics/, 2010

[2] Olmr, Vít, *HW server představuje: Paralelní port - LPT (IEEE 1284)*, http://hw.cz/lpt 2010

[3] Olivka, Petr, CVICENÍ ARP – Úvod *http://poli.cs.vsb.cz/edu/arp/down/arp-cvic.pdf 2010*

[4] http://sourceforge.net/apps/trac/scummvm/browser/vendor/freesci/glutton/src/win32/usleep.c 2010

9. Seznam obrázků

10. Seznam použitých výpisů

Obsah CD

Adresář	**Popis**
Software	implementovaný software
Text	text bakalářské práce
Zdroje	použité elektronické zdroje